Module2: Problem Solving

Source Book

Stuart J. Russell and Peter Norvig, “Artificial Intelligence”, 3rd Edition, Pearson,2015

Topics

1. **Agents, The structure of agents.**
2. **Problem Solving Agents**
3. **Example problems,**
4. **Searching for Solutions,**
5. **Uninformed Search Strategies:**
   a) **Breadth First search,**
   b) **Depth First Search**
   c) **Iterative Deepening Depth First Search**

What is AI?

- Artificial Intelligence (AI) is a field of computer science dedicated to develop systems capable of performing tasks that would typically require human intelligence.
  - These tasks include
    - ***perception,***
    - ***Reasoning/problem solving,***
    - ***learning,***
    - ***understanding natural language**, and*
    - ***interacting with the environment.***
- **According to Russell and Norvig, AI can be defined as follows:**
  - " *AI (Artificial Intelligence) is the study of agents that **perceive their environment, reason about it, and take actions to achieve goals***."

Agent

An agent is defined as anything capable of perceiving its environment through sensors and acting upon that environment through actuators. This basic concept is depicted in Figure 2.1.

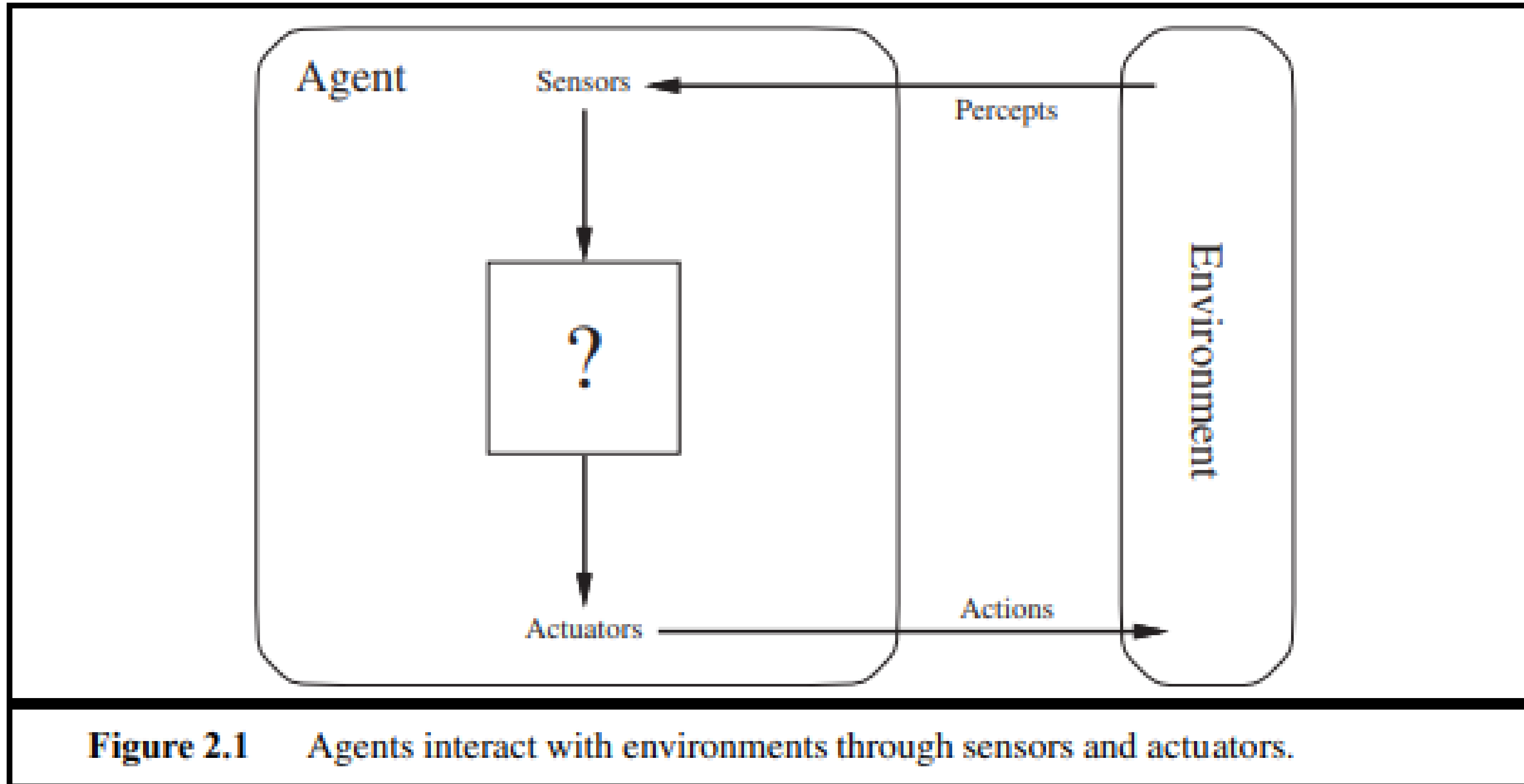


Figure 2.1 Agents interact with environments through sensors and actuators.

Structure of Intelligent Agents

Agent = Architecture + Agent Program

- **Architecture = the machinery that an agent executes on.**
- **Agent Program = an implementation of an agent function.**

Types of Agents

Agent Type	Description
Simple Reflex Agents	Select actions based on the current percept, without considering the history of past percepts.
Model-Based Reflex Agents	Maintain an internal model of the world, considering the history of percepts for decision-making.
Goal-Based Agents	Designed to achieve specific objectives, using internal goals to determine actions.
Utility-Based Agents	Evaluate actions based on a utility function, quantifying the desirability of different outcomes.
Learning Agents	Improve performance over time through learning from experience.

Problem Solving Agents

Problem Solving Agent is a type of goal based intelligent agent in artificial intelligence that is designed to

- **analyse a situation,**
- **identify problems or goals, and then**
- **take actions to achieve those goals**.

Problem Solving Agents

Problem Solving Agent is a type of goal based intelligent agent in artificial intelligence that is designed to **analyse a situation, identify problems or goals, and then take actions to achieve those goals**.

- In the city of **Arad**, Romania, an agent on a touring holiday has a performance measure with various goals, **such as improving suntan, language skills, exploring sights, and avoiding hangovers.**
- The decision problem is complex if no goal is fixed.

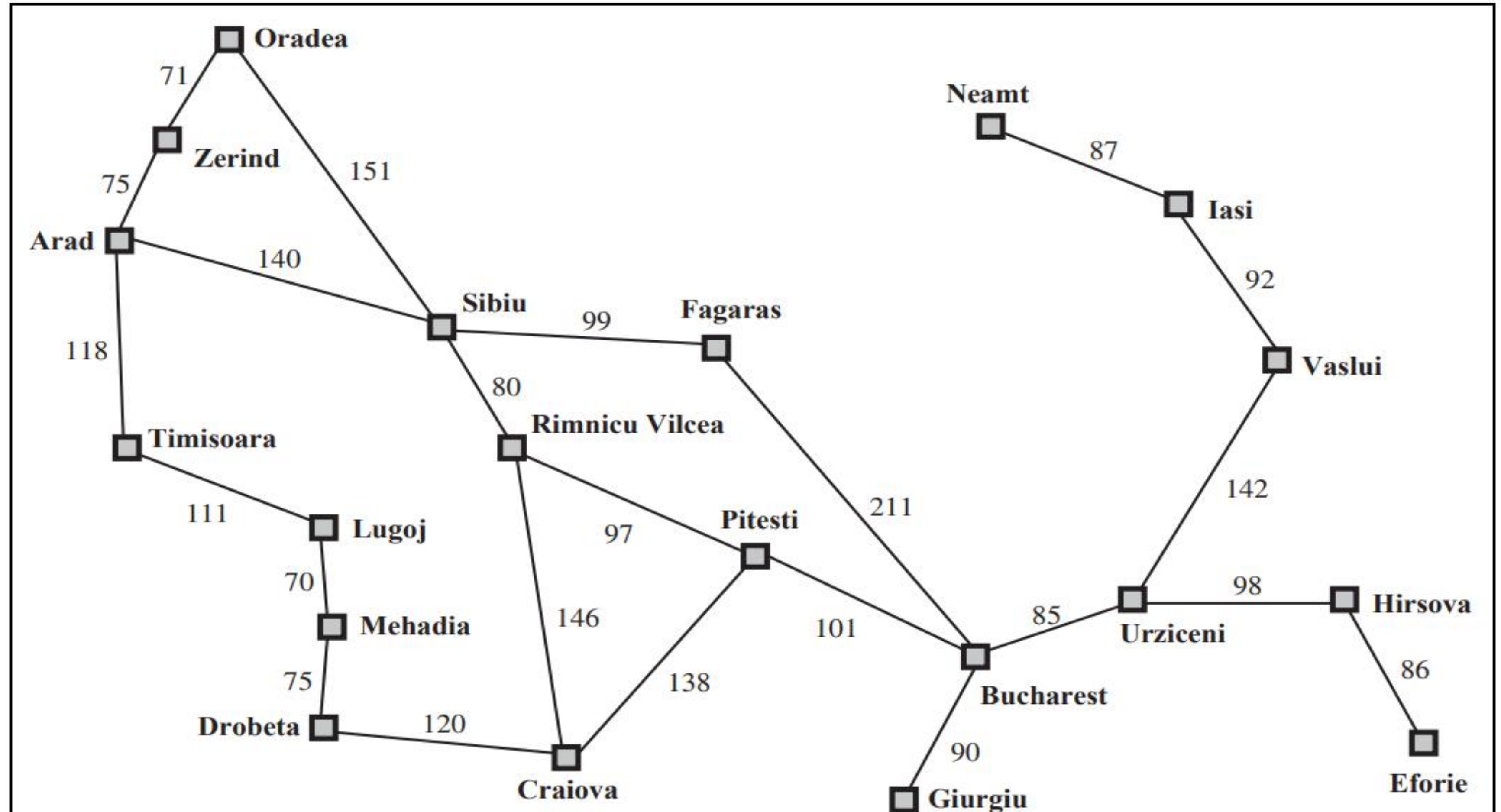


Figure 3.2 A simplified road map of part of Romania.

Steps followed by Problem Solving Agents

1. **Goal Formulation**
2. **Problem Formulation**
3. **Search Solution**
4. **Execution**
5. **Learning (Optional)**
6. **Feedback and Iteration**

"Formulate, Search, Execute" framework for the agent

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Figure 3.1 A simple problem-solving agent. It first formulates a goal and a problem, searches for a sequence of actions that would solve the problem, and then executes the actions one at a time. When this is complete, it formulates another goal and starts over.

Well Defined Problems and Solutions

A problem can be defined formally by five components:

1. **State Representation**: Encompasses the initial state from which the agent begins its problem-solving journey, represented, for example, as "*In(Arad)*."
2. **Actions and Applicability**: Describes the set of possible actions available to the agent in a given state, denoted as ACTIONS(s). For instance, in the state *In(Arad)*, applicable actions include {*Go(Sibiu), Go(Timisoara), Go(Zerind)}*.
3. **Transition Model**: Specifies the consequences of actions through the transition model, represented by the function RESULT(s,a), which yields the state resulting from performing action a in state s. For example, RESULT(*In(Arad),Go(Zerind))=In(Zerind)*.
4. **Goal Specification and Test**: Defines the goal state or states and includes a test to determine whether a given state satisfies the goal conditions. In the example, the goal is represented as the singleton set *{In(Bucharest)}*.
5. **Cost Functions**: Encompasses both the path cost function, assigning a numeric cost to each path, and the step cost, denoted as $c(s,a,s')$, which represents the cost of taking action **a** in state **s** to reach state s'. The cost functions play a crucial role in evaluating and optimizing the performance of the agent's solution.

Example Problems: Toy and Real-world problems.

A toy problem is designed to showcase or test various problem-solving techniques, featuring a precise and concise description. This allows different researchers to use it for comparing algorithm performances.

On the other hand, **a real-world problem** is one that holds significance for people, lacking a single universally agreed-upon description. However, we can provide a general sense of their formulations.

Toy Problems

1. **Vacuum Cleaner World**
2. **8 Puzzle**
3. **8 Queens**
4. **Math's Sequences**

Vacuum Cleaner World

The problem can be formalized as follows:

1. **States:** The state is defined by the agent's location and the presence of dirt in specific locations. The agent can be in one of two locations, each potentially containing dirt. Consequently, there are 8 possible world states (2×2^2). For a larger environment with n locations, there would be $\mathbf{n \cdot 2^n}$ states.
2. **Initial state**: Any state can serve as the initial state.
3. **Actions:** In this uncomplicated environment, each state presents three actions: **Left, Right, and Suck**. More extensive environments might also include **Up** and **Down**.
4. **Transition model**: Actions produce expected effects, except for instances where moving **Left** in the leftmost square, moving **Right** in the rightmost square, and **Sucking in a clean square** result in no effect.
5. **Goal test**: This assesses whether **all squares are clean**.
6. **Path cost**: Each step incurs a **cost of 1**, making the path cost equivalent to the number of steps taken in the path.

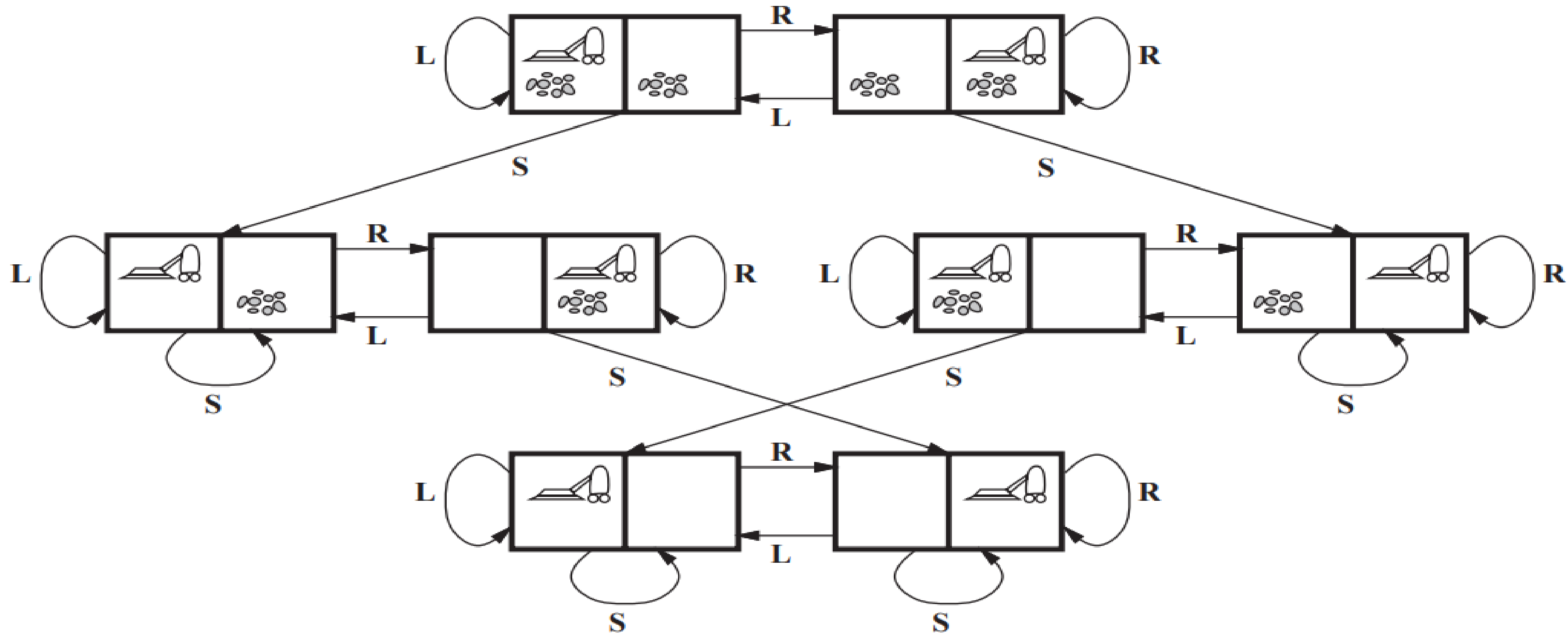


Figure 3.3 The state space for the vacuum world. Links denote actions: L = *Left*, R = *Right*, S = *Suck*.

8 Puzzle

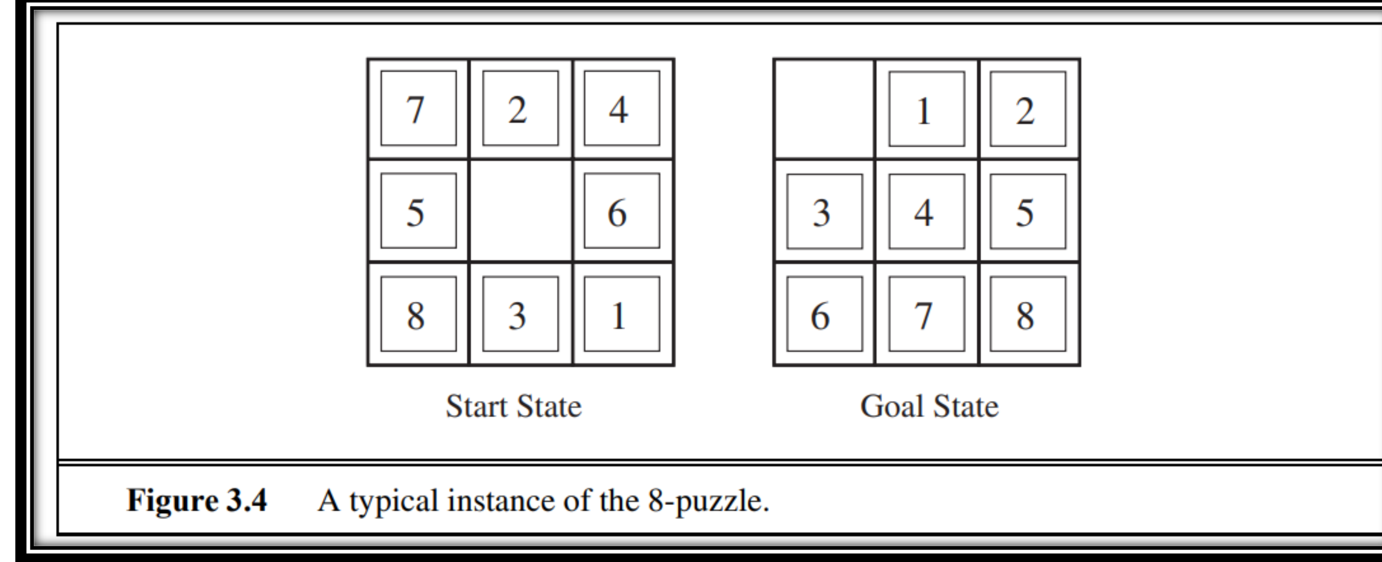
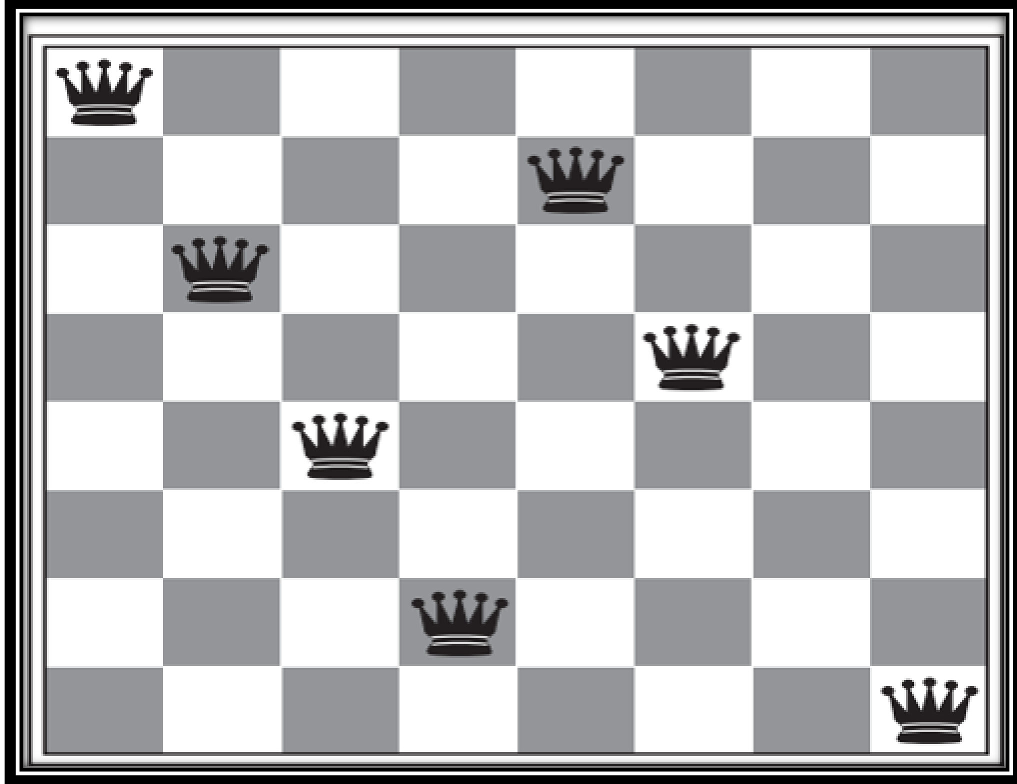


Figure 3.4 A typical instance of the 8-puzzle.

1. **States:** A state description indicates the position of each of the eight tiles and the empty space within the nine squares.
2. **Initial state:** Any state can be designated as the initial state.
3. **Actions:** In its simplest form, actions are defined as movements of the empty space—Left, Right, Up, or Down. Different subsets of these actions are possible based on the current location of the empty space.
4. **Transition model**: Given a state and an action, the model returns the resulting state. For instance, applying Left to the starting state in Figure 3.4 would switch the positions of the 5 and the empty space.
5. **Goal test:** This checks if the state aligns with the specified goal configuration shown in Figure 3.4. Other goal configurations are also conceivable.
6. **Path cost:** Each step incurs a cost of 1, making the path cost equivalent to the number of steps taken in the path.

The 8-queens problem



- The goal of the 8-queens problem is to place eight queens on a chessboard such that no queen attacks any other.
- (A queen attacks any piece in the same row, column or diagonal.)
- Figure 3.5 shows an attempted solution that fails: the queen in the rightmost column is attacked by the queen at the top left.

- **States:** Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state:** No queens on the board.
- **Actions:** Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

Math's Sequences

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5 .$$

The problem definition is very simple:

1. **States**: Positive numbers.
2. **Initial state**: 4.
3. **Actions**: Apply factorial, square root, or floor operation (factorial for integers only).
4. **Transition model**: As given by the mathematical definitions of the operations.
5. **Goal test**: State is the desired positive integer

Real-world problems

1. Route Finding Problem
2. Touring Problem
3. Traveling Salesperson Problem
4. VLSI Layout
5. Robot Navigation

Consider the airline travel problems that must be solved by a travel-planning Web site:

- **States:** Each state obviously includes **a location (e.g., an airport**) and the **current time**. Furthermore, because the **cost of an action (a flight segment)** may depend on previous segments, **their fare bases**, and their status as **domestic or international**, the state must record extra information about **these "historical" aspects**.

- **Initial state**: This is specified by the **user's query.**

- **Actions**: **Take any flight from the current location**, in any **seat class**, leaving after the **current time**, leaving enough time for within-airport transfer if needed.

- **Transition model**: The state resulting from taking a flight will have the **flight's destination** as the **current location** and the **flight's arrival time as the current time**.

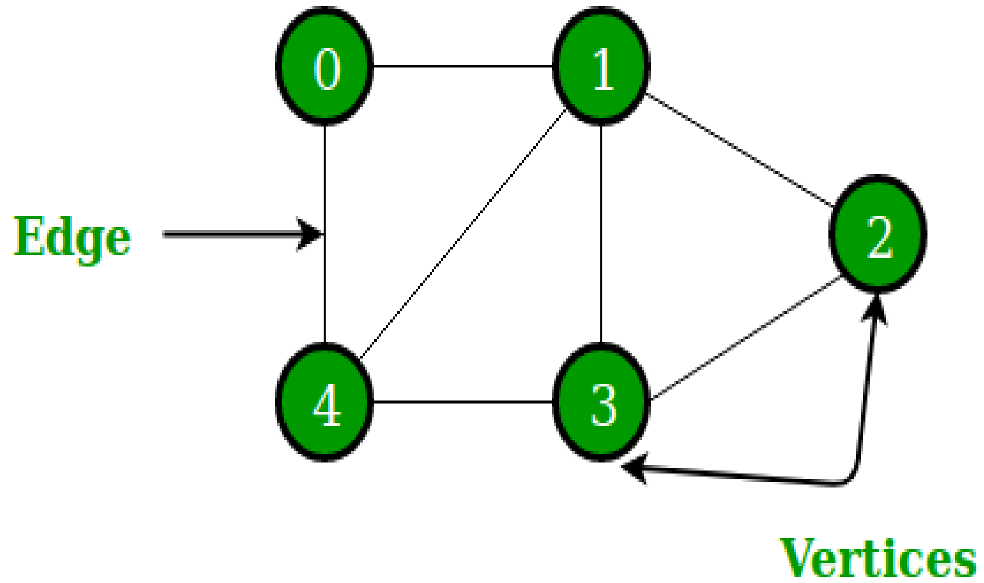
- **Goal test**: Are we at the **final destination** specified by the user?

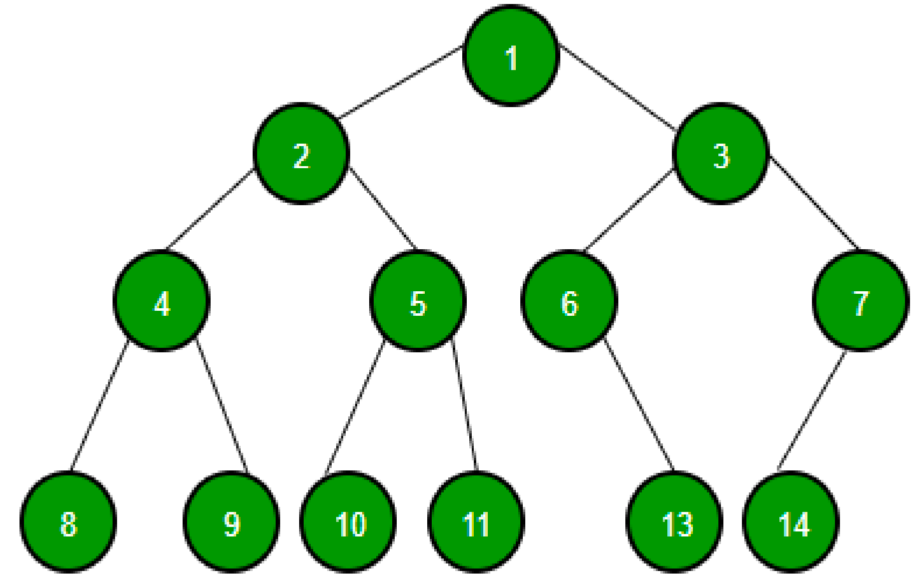
- **Path cost**: This depends on **monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, and so on**.

Parameter	Graph	Tree
Description	Graph is a non−linear data structure that can have more than one path between vertices.	Tree is also a non−linear data structure, but it has only one path between two vertices.
Loops	Graphs can have loops.	Loops are not allowed in a tree structure.
Root Node	Graphs do not have a root node.	Trees have exactly one root node.
Traversal Techniques	Graphs have two traversal techniques namely, breadth−first search and depth−first search.	Trees have three traversal techniques namely, pre−order, in−order, and post−order.

Graph

Trees





Searching for Solutions

- The **SEARCH TREE** possible action sequences starting at the initial state form a search tree with the initial state NODE at the root; the branches are actions and the nodes correspond to states in the state space of the problem
- **Expanding** the current state applying each legal action to the current state, thereby **generating** a new set of state.

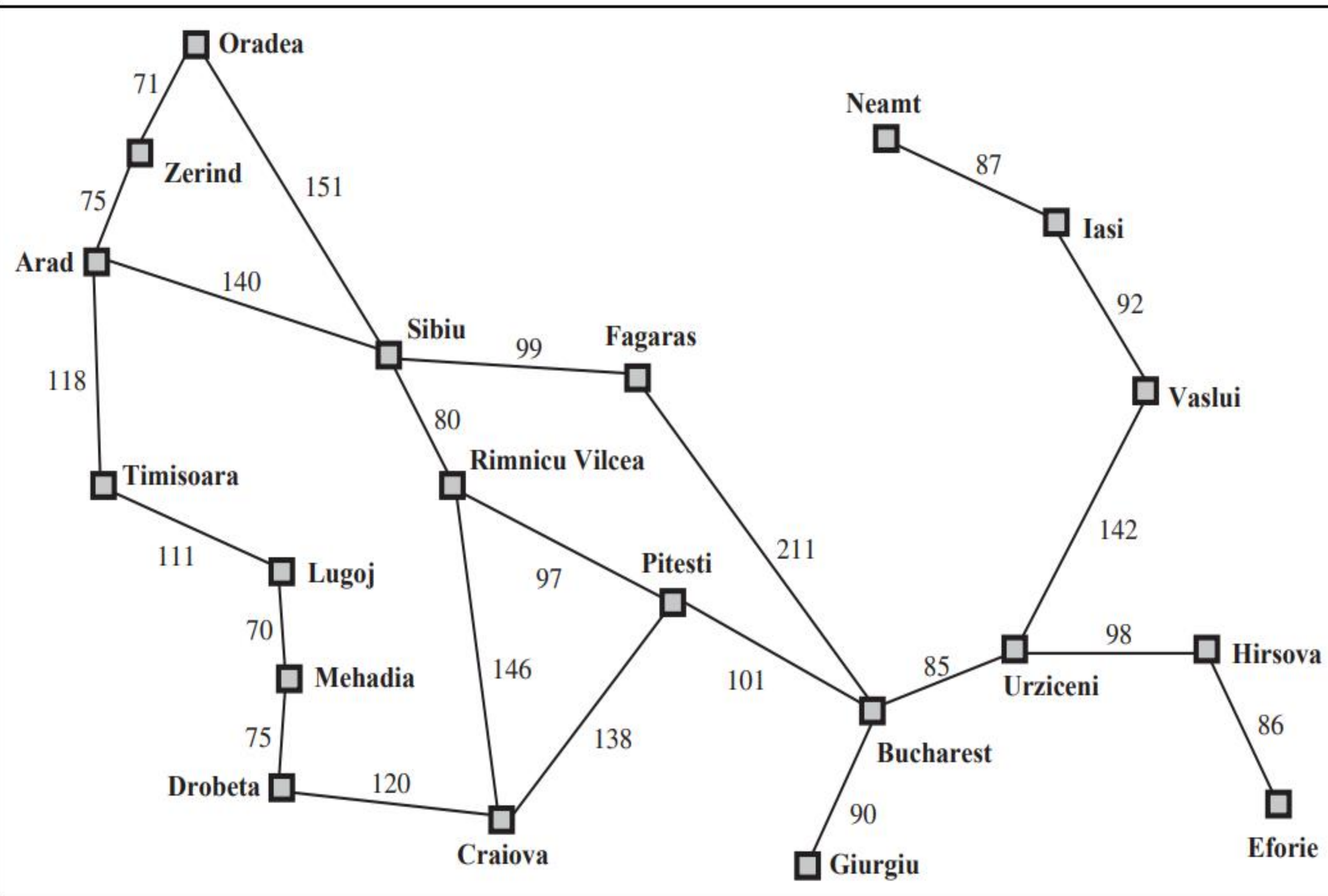
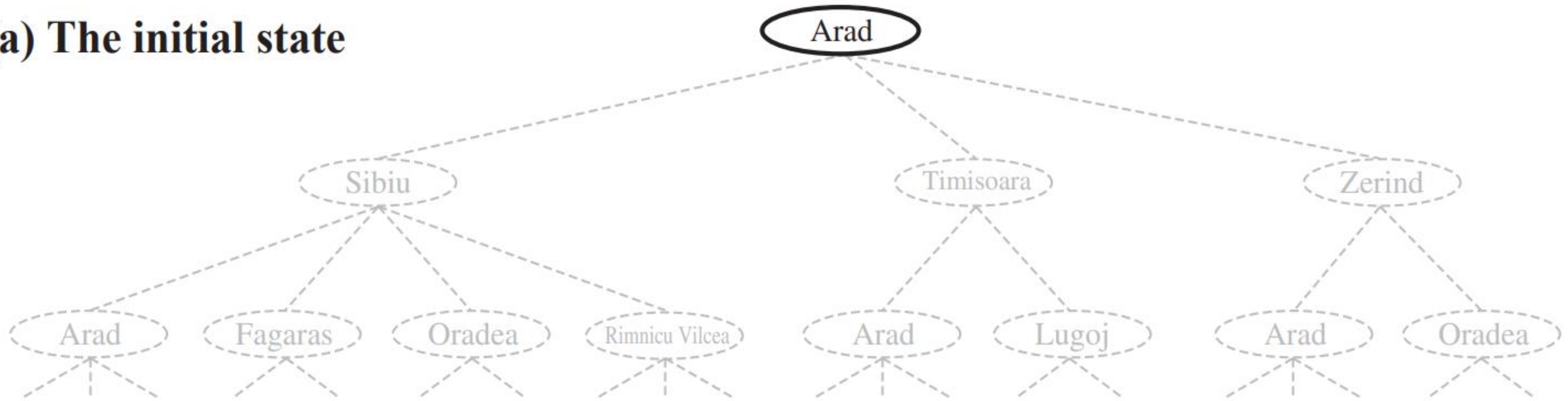


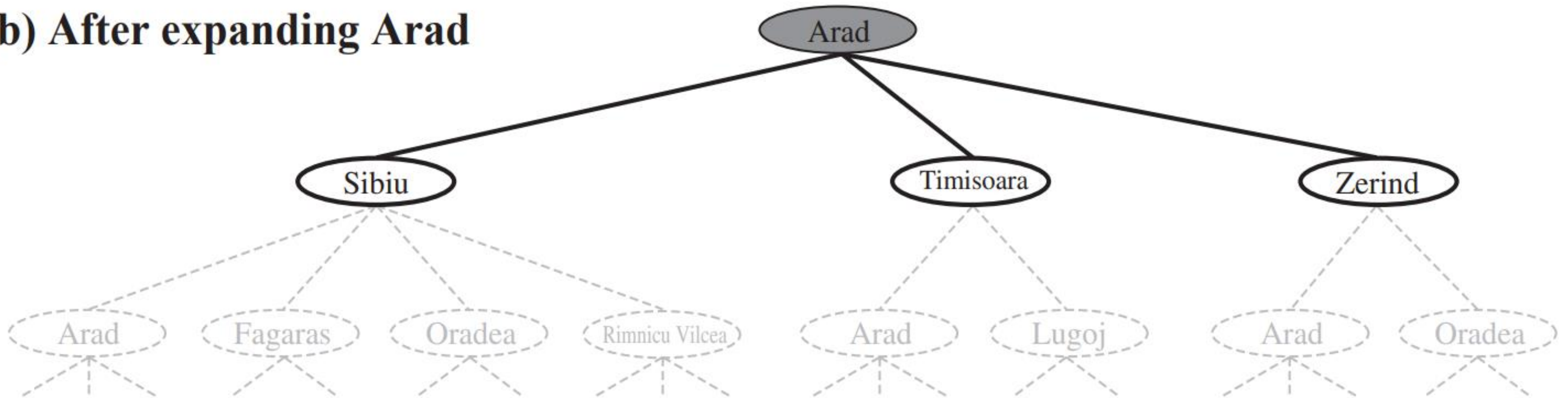
Figure 3.2 A simplified road map of part of Romania.

[Source Book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015]

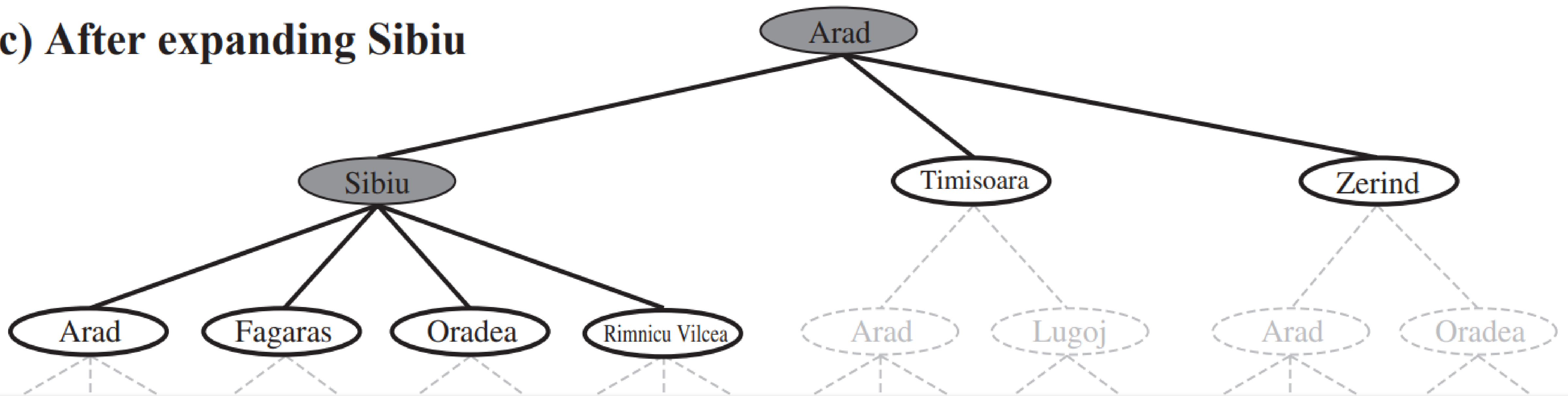
Partial search trees for finding a route from Arad to Bucharest



Partial search trees for finding a route from Arad to Bucharest



Partial search trees for finding a route from Arad to Bucharest



Tree Based Searching

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     expand the chosen node, adding the resulting nodes to the frontier
```

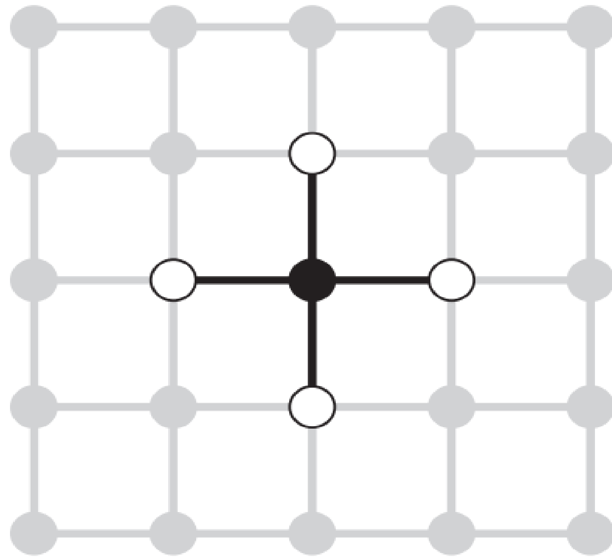
Graph based Searching

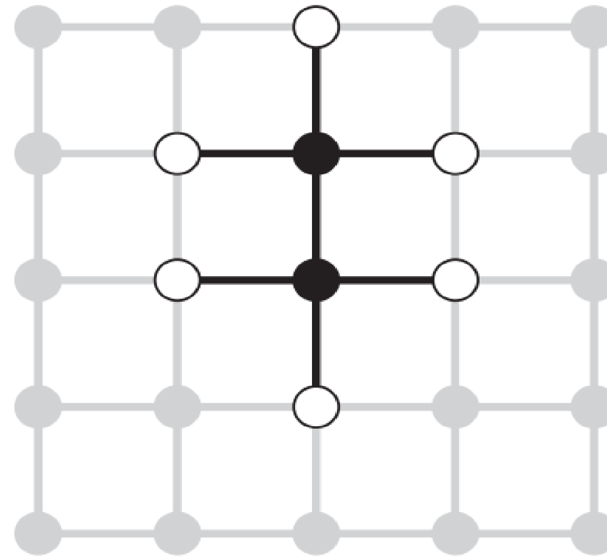
```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
     if the frontier is empty then return failure
     choose a leaf node and remove it from the frontier
     if the node contains a goal state then return the corresponding solution
     add the node to the explored set
     expand the chosen node, adding the resulting nodes to the frontier
        only if not in the frontier or explored set
```

Sequence of Search Trees

Figure 3.8 A sequence of search trees generated by a graph search on the Romania problem of Figure 3.2. At each stage, we have extended each path by one step. Notice that at the third stage, the northernmost city (Oradea) has become a dead end: both of its successors are already explored via other paths.

Graph Search


(a)


(b)

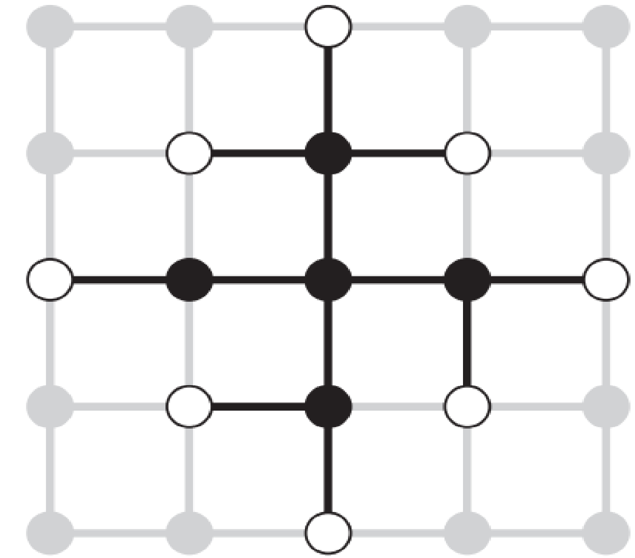

(c)

Figure 3.9 The separation property of GRAPH-SEARCH, illustrated on a rectangular-grid problem. The frontier (white nodes) always separates the explored region of the state space (black nodes) from the unexplored region (gray nodes). In (a), just the root has been expanded. In (b), one leaf node has been expanded. In (c), the remaining successors of the root have been expanded in clockwise order.

Infrastructure for search algorithms

Search algorithms require a data structure to keep track of the search tree that is being constructed. For each **node n** of the tree, we have a structure that contains four components:

- **n.STATE**: the state in the state space to which the node corresponds;
- **n.PARENT**: the node in the search tree that generated this node;
- **n.ACTION**: the action that was applied to the parent to generate the node;
- **n.PATH-COST**: the cost, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers.

NODE

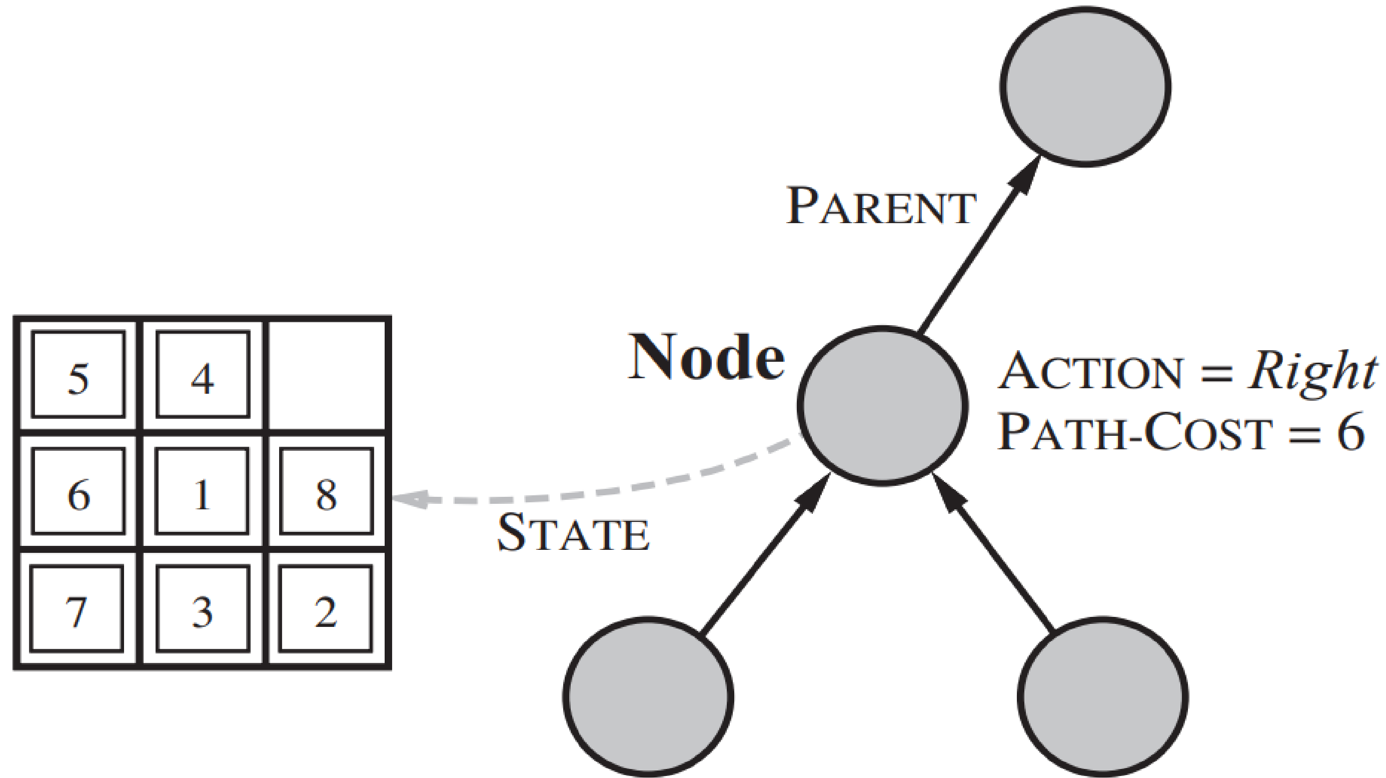


Figure 3.10 Nodes are the data structures from which the search tree is constructed. Each has a parent, a state, and various bookkeeping fields. Arrows point from child to parent.

Function: "CHILD NODE"

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Measuring problem-solving performance

We can evaluate an algorithm’s performance in four ways:

1. **COMPLETENESS** : Is the algorithm guaranteed to find a solution when there is one?
2. **OPTIMALITY** : Does the strategy find the optimal solution?
3. **TIME COMPLEXITY** : How long does it take to find a solution?
4. **SPACE COMPLEXITY** : How much memory is needed to perform the search?

Searching Strategies/Algorithms

Uninformed (Blind)Search Strategies

1. Breadth-first search
2. Uniform-cost search
3. Depth-first search
4. Depth-limited search
5. Iterative deepening depth-first search
6. Bidirectional search

Informed (Heuristic) Search Strategies

1. Greedy best-first search
2. A* search: Minimizing the total estimated solution cost
3. Memory-bounded heuristic search
4. AO* Search
5. Problem Reduction
6. Hill Climbing

Comparing uninformed search strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

Breadth First Search

1. **Initialize a queue with the initial state (usually the root node).**
2. **While the queue is not empty:**
   a. **Dequeue a node from the front of the queue.**
   b. **If the node contains the goal state, return the solution.**
   c. **Otherwise, enqueue all the neighbouring nodes that have not been visited.**
3. **If the queue becomes empty and the goal state is not found, then there is no solution.**

Note : The set of all leaf nodes available for expansion at any given point is called the frontier

Pseudo Code

```
function BFS(initial_state, goal_state):
    initialize an empty queue
    enqueue initial_state to the queue

    while queue is not empty:
        current_node = dequeue from the front of the queue

        if current_node is the goal_state:
            return solution
        for each neighbor of current_node:
            if neighbor has not been visited:
                mark neighbor as visited
                enqueue neighbor to the queue

return no solution
```

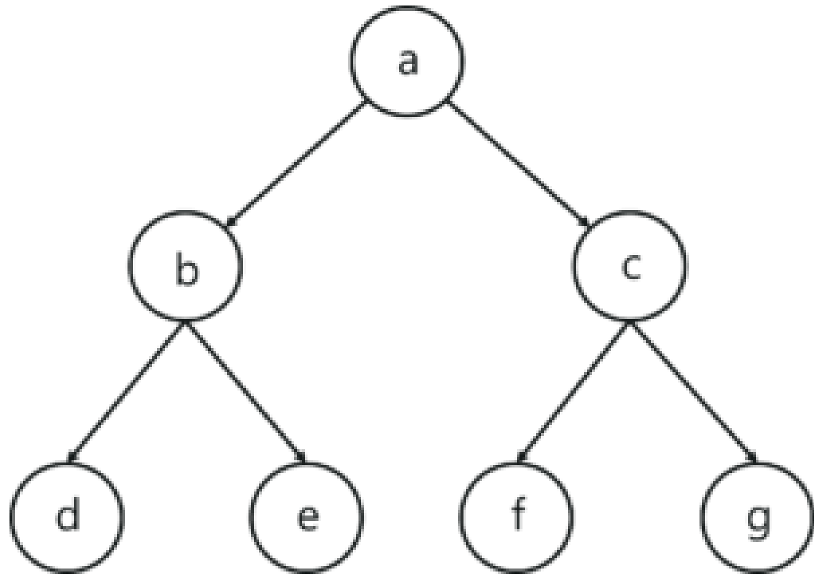
Note:

- The queue is a **First-In-First-Out (FIFO)** data structure, meaning that the first element enqueued will be the first to be dequeued.
- The algorithm ensures that all nodes at a particular depth level are explored before moving on to the nodes at the next level.
- BFS is complete and optimal for searching in a state space with a uniform cost per step.
- It may require a lot of memory for large state spaces due to the need to store all generated nodes.

Example

Step	Details	Visited Nodes	QUEUE
1	Initialization: Start with the root node A.	V = {}	a
2	Node **a,** Visited, Dequeue node **a** and enqueue neighbour nodes **b** and **c** to queue	V = {a}	(empty) .. b / c b
3	Node **b** visited, deque node **b** and enqueue neighbour nodes **d** and **e** to queue.	V = {a,b}	c .. d c / e d c
4	Node **c** visited, deque node **c** and enqueue neighbour nodes **f** and **g** to queue	V = {a,b,c}	e d / f e d / g f e d

Example



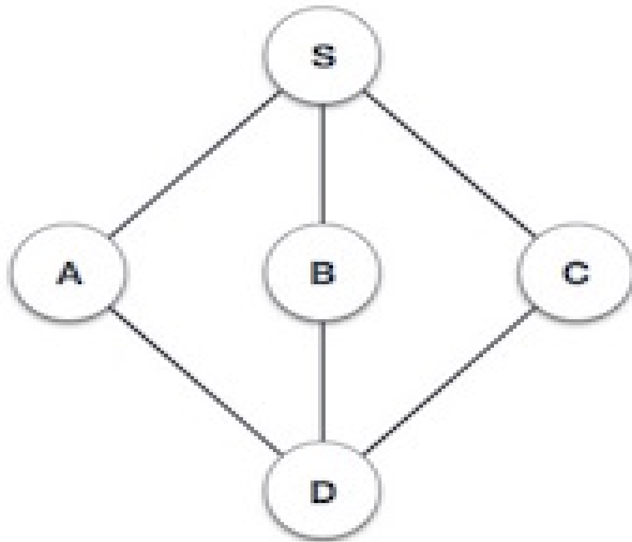
5	Node **d** visited, deque node **d**	V = {a,b,c,d}	g f e
6	Node **e** visited, deque node **e**	V = {a,b,c,d,e}	g f
7	Node **f** visited, deque node **f**	V = {a,b,c,d,e,f}	g
8	Node **g** visited, deque node **g**	V = {a,b,c,d,e,f,g}	

Pseudo Code BFS

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     add node.STATE to explored
     for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

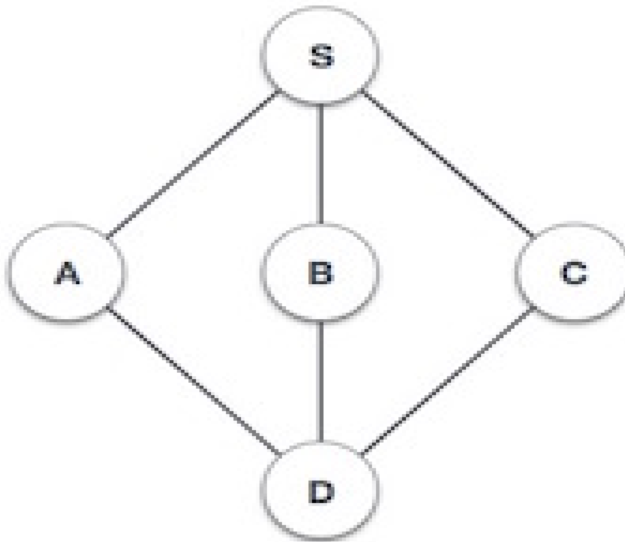
Figure 3.11 Breadth-first search on a graph.

Example: BFS on Simple Graph



Step	Description	Visited Nodes	Queue
1	Initialize the queue.	V= {}	S
2	Node **S** Visited, Dequeue node **S** and enqueue neighbour nodes **A, B** and **C** to queue.	V= {S}	A B A C B A
3	Node **A** Visited, Dequeue node **A** and enqueue neighbour node **D**	V= {S,A}	D C B

Example: BFS on Simple Graph



4	Node **B** Visited, Dequeue node **B**	V= {S,A,B}	D C
5	Node **C** Visited, Dequeue node **C**	V= {S,A,B,C}	D
6	Node **D** Visited, Dequeue node **D**	V= {S,A,B,C,D}	

Breadth First Search: S A B C D

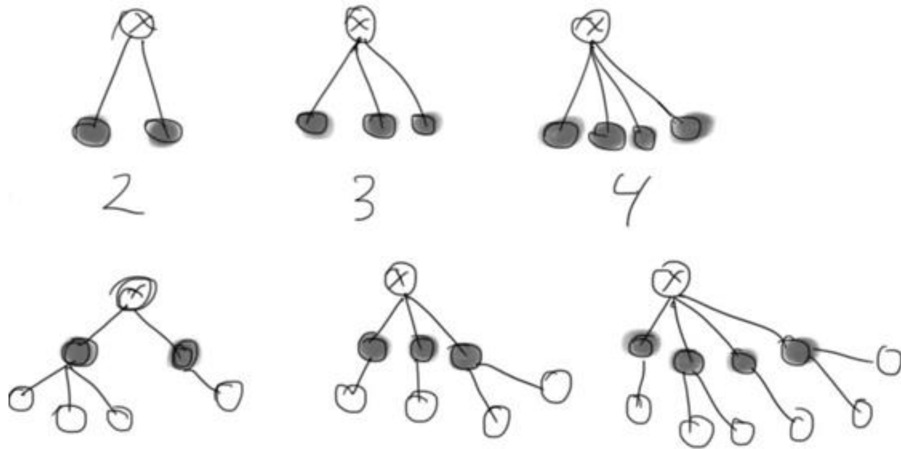
Time and Space Complexity of BFS

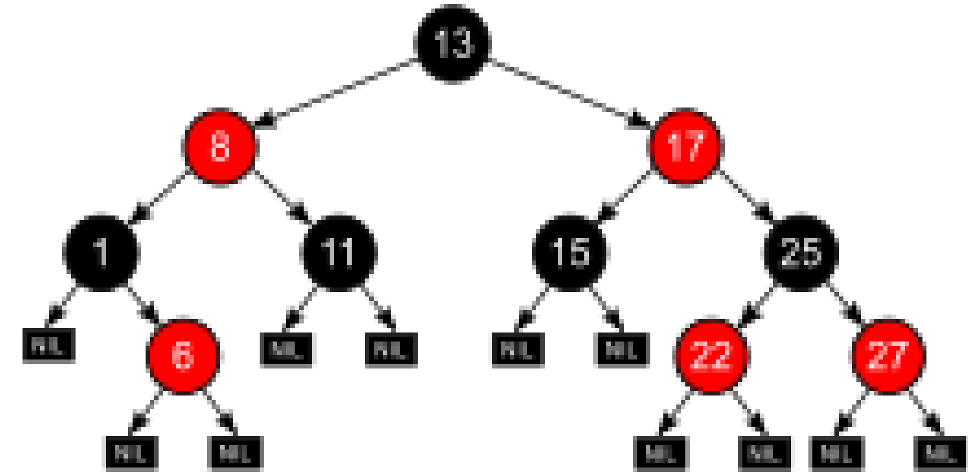
Time Complexity:

- The time complexity of BFS in a uniform tree is expressed as $O(b^d)$, where 'b' is the branching factor and 'd' is the depth of the solution.
- Applying the goal test upon node expansion instead of generation would result in a higher time complexity of $O(b^{(d+1)})$.

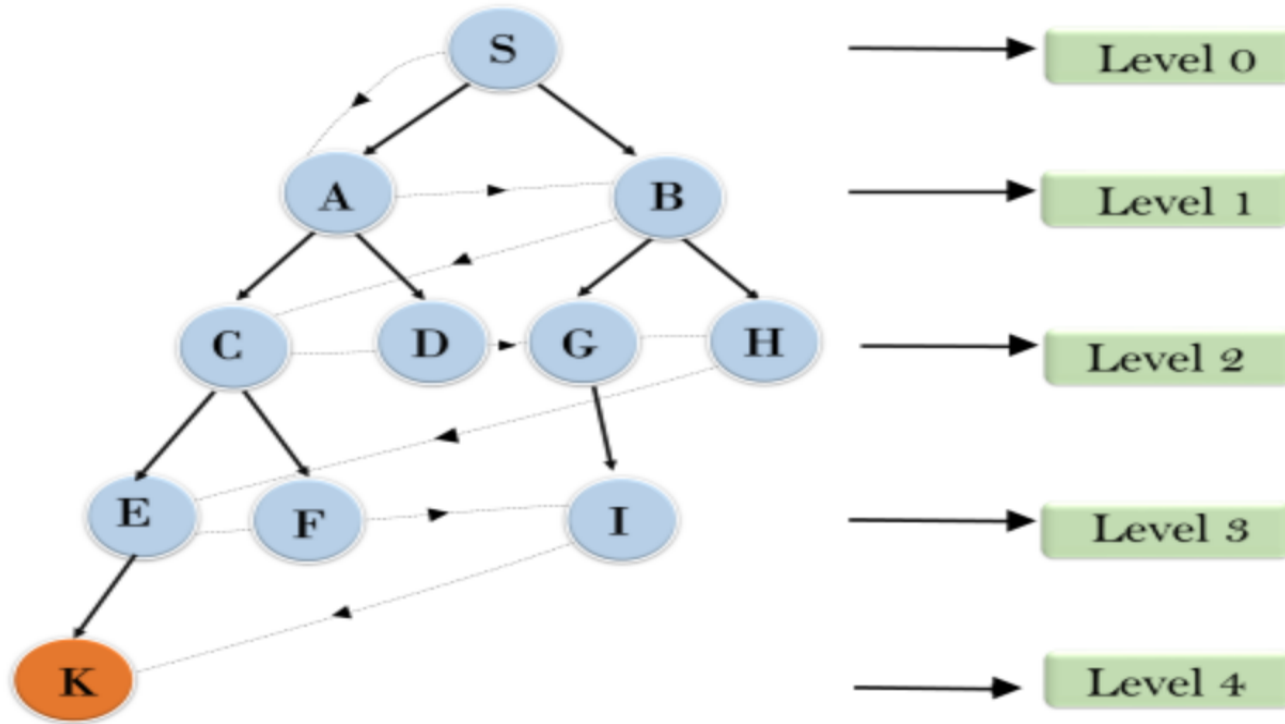
Space Complexity:

- For breadth-first graph search, where every generated node is kept in memory, the space complexity is $O(b^d)$.
- The space complexity is dominated by the size of the frontier, which holds nodes yet to be explored.





- The **branching factor** is the number of children at each node, the outdegree. If this value is not uniform, an *average branching factor* can be calculated.
- The average branching factor can be quickly calculated as the number of non-root nodes (the size of the tree, minus one; or the number of edges) divided by the number of non-leaf nodes (the number of nodes with children)



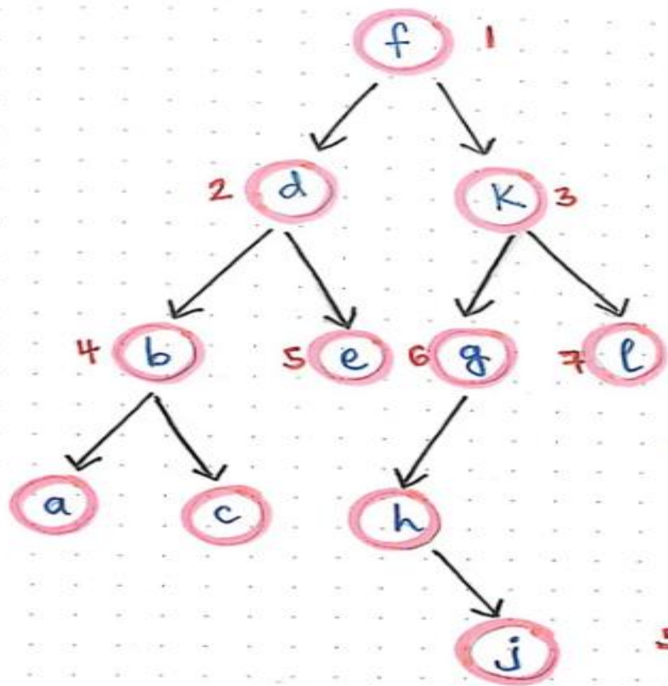
e
a
n

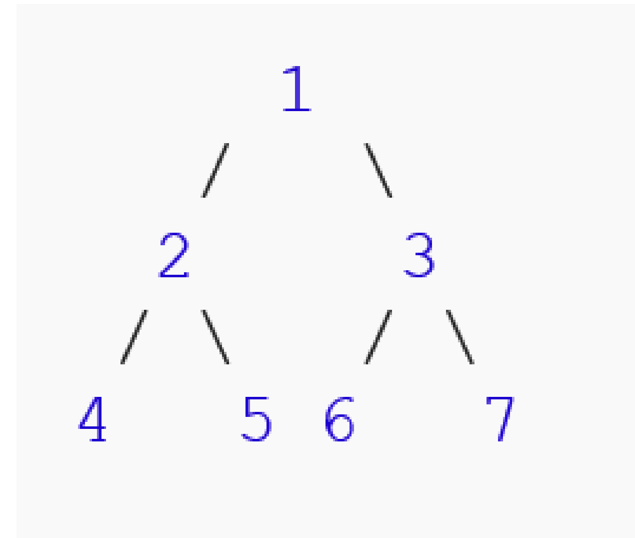
Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

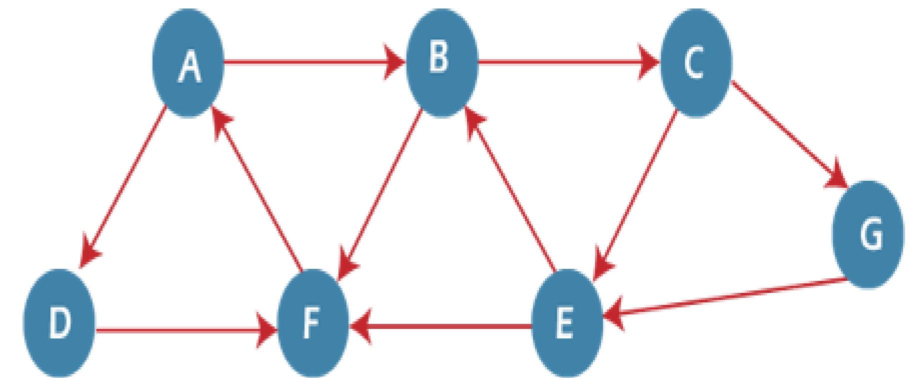
T (b) = $1+b^2+b^3+\ldots\ldots+b^d= O(b^d)$

T= 1 + 2 + 4 + 8 + 16 = 31

Exercise: Apply BFS on Following







Depth First Search

- Depth-First Search (DFS) is a tree traversal algorithm that explores as far as possible along each branch before backtracking.
- In the context of a binary tree, DFS can be implemented using **recursion or a Stack (LIFO QUEUE)**. Let's go through the DFS algorithm on a binary tree with an example.

DFS Algorithm on Binary Tree:

Recursive Approach:

- Start at the root node.
- Visit the current node.
  - Recursively apply DFS to the left subtree.
  - Recursively apply DFS to the right subtree.

Stack-Based Approach:

- Push the root node onto the stack.
- While the stack is not empty:
  - Pop a node from the stack.
  - Visit the popped node.
  - Push the right child onto the stack (if exists).
  - Push the left child onto the stack (if exists).

DFS Algorithm with LIFO Queue:

Initialization:

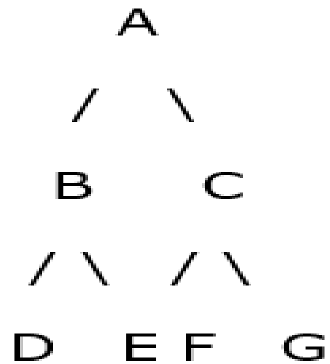
Push the root node onto the LIFO queue.

Traversal:

While the LIFO queue is not empty:

- Pop a node from the LIFO queue.
- Visit the popped node.
- Push the right child onto the LIFO queue (if exists).
- Push the left child onto the LIFO queue (if exists).

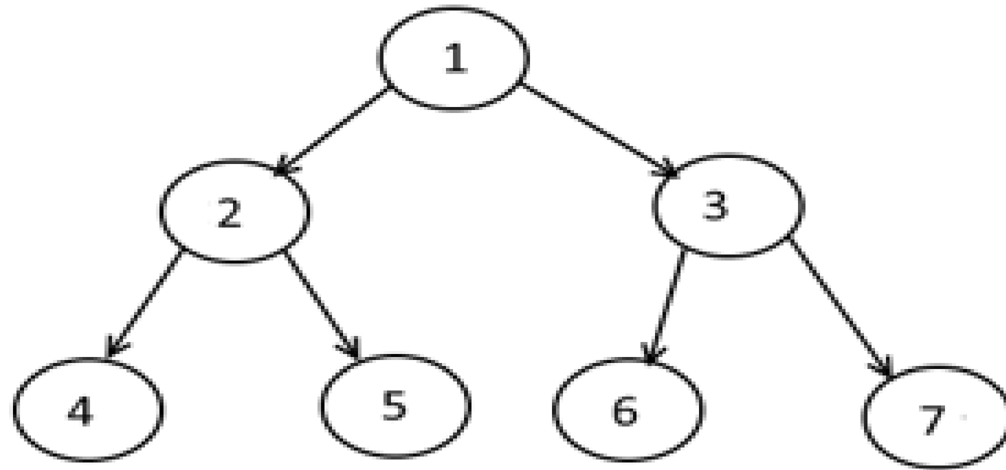
Example



Step	Description	Visited Nodes	LIFO QUEUE
1	Push 'A' onto the LIFO queue	{}	[A]
2	Pop 'A', visit it, and push its children 'B' and 'C' onto the LIFO queue	{A}	[C, B]
3	Pop 'B', visit it, and push its children 'E' and 'D' onto the LIFO queue.	{A, B}	[C, E, D]
4	Pop 'D', visit it (no children to push)	{A,B,D}	[C, E]
5	Pop 'E', visit it (no children to push)	{A,B,D,E}	[C]
6	Pop 'C', visit it, and push its children 'F' and 'G'	{A,B,D,E,C}	[G,F]
7	Pop 'F', visit it (no children to push)	{A,B,D,E,C,F}	[G]
8	Pop 'G', visit it (no children to push)	{A,B,D,E,C,F,G}	[]

Resulting DFS Traversal Order: **A,B,D,E,C,F,G**

Exercise: Apply DFS for the Following



Discuss the Time and Space Complexity of DFS

DFS and BFS Time and Space Complexity

Strategy	Graph		Binary Tree	
	Time Complexity	Space Complexity	Time Complexity	Space Complexity
BFS	**O(V + E)**	**O(V)**	$O(b^d)$	$O(b^d)$
DFS	**O(V + E)**	**O(V)**	$O(b^d)$	O(bd)

Depth Limited Search

Algorithm:

Initialization:

- Set the depth limit to l
- Push the root node onto the LIFO queue.

Traversal:

d = 0

While the LIFO queue is not empty and d<=l and goal is not reached:

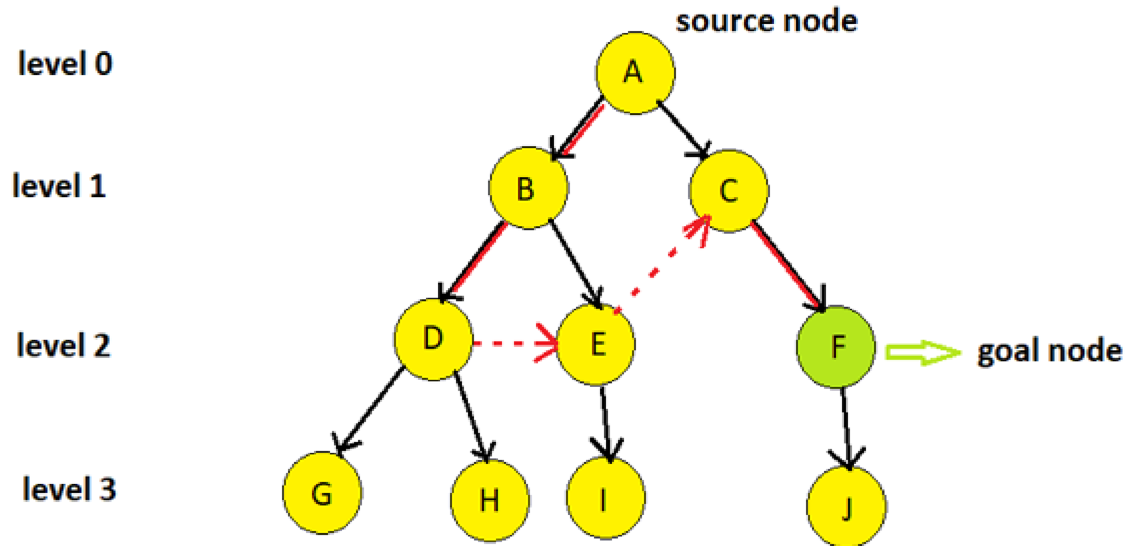
- Pop a node from the LIFO queue.
- Visit the popped node.
- Push the **right child** onto the LIFO queue (if exists).
- Push the **left child** onto the LIFO queue (if exists).
- Update the depth if goal is not reached

Example

```
      1
     / \
    2   3
   / \
  4   5
 / \
6   7
```

Step	Description	Visited Nodes	LIFO QUEUE	Depth d
1	Push '1' onto the LIFO queue	{}	[1]	0 < 2
2	Pop '1', visit it, and push its children '2' and '3' onto the LIFO queue	{1}	[3,2]	1<2
3	Pop '2', visit it, and push its children '5' and '4' onto the LIFO queue.	{1, 2}	[3,5,4]	2=2
4	Pop '4', visit it (no children to push)	{1,2,4}	[3, 5]	2
5	Pop '5', visit it (Goal reached)	{1,2,4,5}	[3]	2

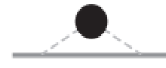
Example



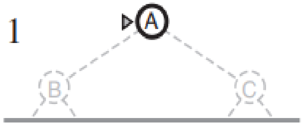
Iterative Deepening Search

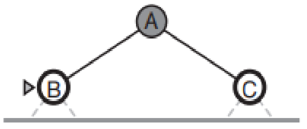
- **Iterative Deepening Depth-First Search (IDDFS)** is a search strategy that combines the benefits of depth-first search and breadth-first search.
- Iterative deepening search, also known as iterative deepening depth-first search.
- This method incrementally raises the limit, starting from 0 and progressing to 1, 2, and so forth, until a goal state is reached, typically when the depth limit attains the value of 'd,' representing the depth of the shallowest goal node.

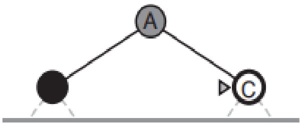
Limit = 0

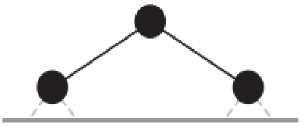


Limit = 1

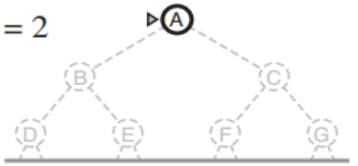


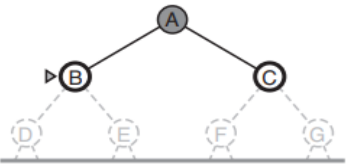


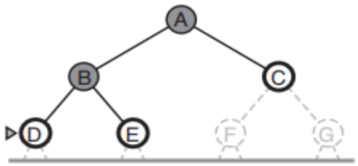


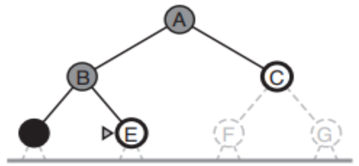


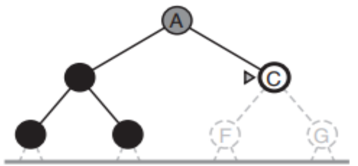
Limit = 2

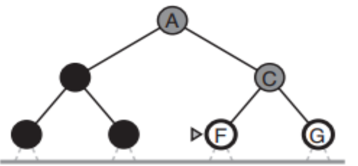


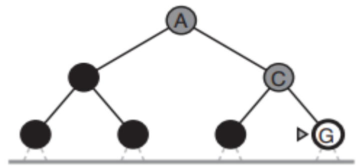


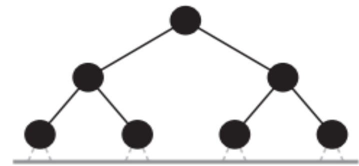


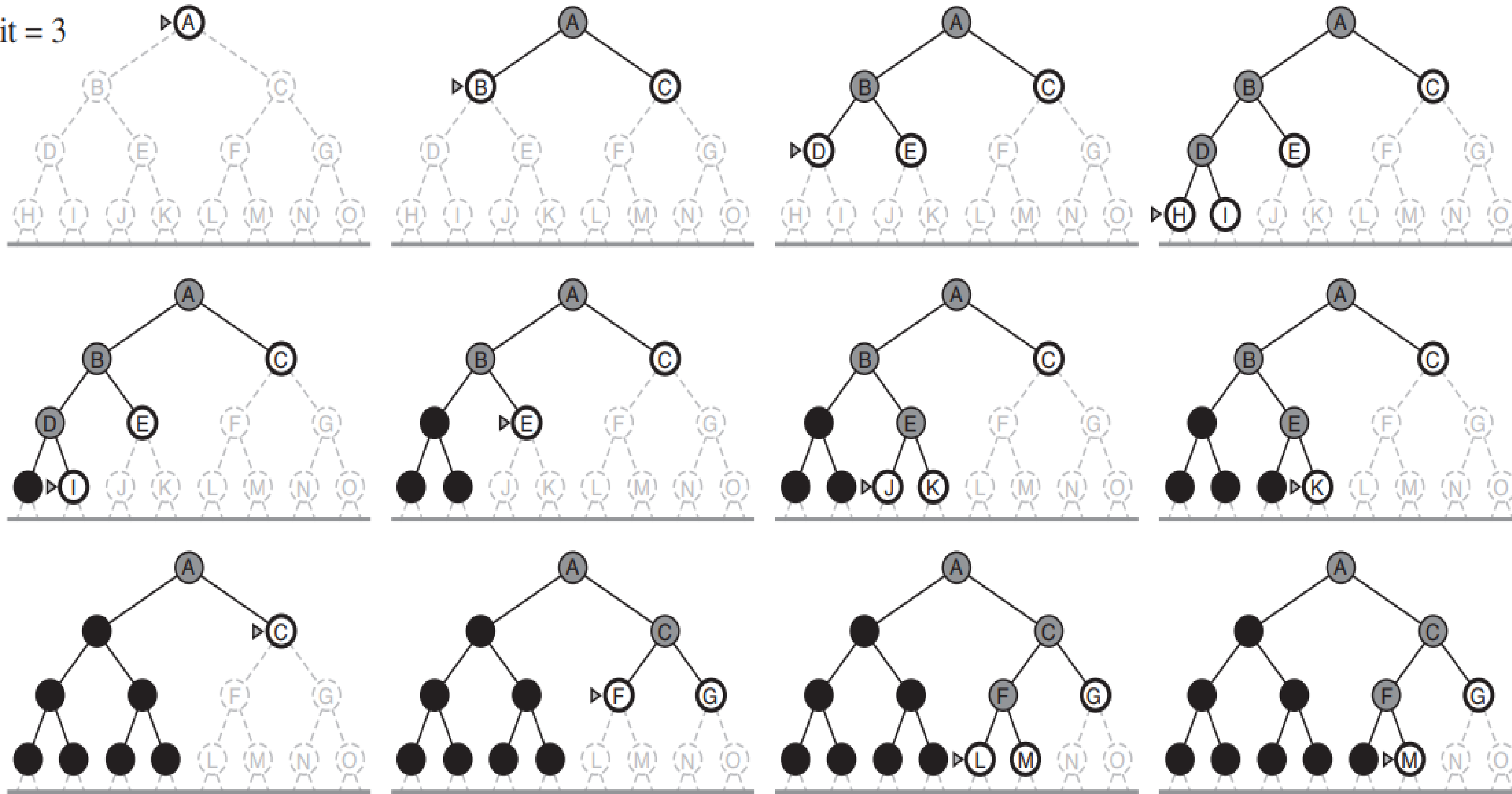








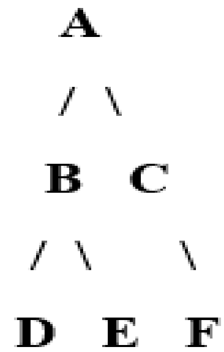




Dr.Thyagaraju G S , Professor,Dept of CSE,SDMIT,Ujire -574240
[Source Book: Stuart J. Russell and Peter Norvig, Artificial Intelligence, 3rd Edition, Pearson,2015]

Example

The goal is to find the node with the value **'F'**. We'll use **Iterative Deepening Depth-First Search** with increasing depth limits.



Iteration 1 (Depth Limit = 0):

- Start at the root node A and perform depth-first search up to depth 0.
- Explore only the root node A.
- No goal found.

Iteration 2 (Depth Limit = 1):

- Start again at the root node A.
- Explore nodes A, B, and C up to depth 1.
- No goal found.

Iteration 3 (Depth Limit = 2):

- Start again at the root node A.
- Explore nodes A, B, C up to depth 2.
- Explore node B's children D and E, and node C's child F.
- Goal 'F' is found.

Time and Space Complexity

Algorithm	Time	Space	Complete	Optimal
Breadth First Search	$O(b^d)$	$O(b^d)$	YES	YES
Depth First Search	$O(b^d)$	O(bd)	NO	NO
Depth Limited Search	$O(b^l)$	O(bl)	NO	NO
Iterative deepening Search	$O(b^d)$	O(bd)	YES	YES

Topics

1. Agents, The structure of agents.
2. Problem Solving Agents
3. Example problems,
4. Searching for Solutions,
5. Uninformed Search Strategies:
   a) Breadth First search,
   b) Depth First Search
   c) Iterative Deepening Depth First Search

END of Module2